Theoretical Computer Science 2 (1976) 183–207.

CORRECTNESS OF PARALLEL PROGRAMS: THE CHURCH–ROSSER APPROACH*

Barry K. ROSEN

Computer Sciences Department, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y. 10598, USA

Communicated by Ronald Book
Received 6 November 1974
Revised 10 June 1975

Abstract. For many purposes, asynchronous parallel programs may be viewed as sequential but nondeterministic programs. The direct translation to nondeterministic sequential form leads to a combinatorial explosion of program size before correctness proofs can even begin.

The Church–Rosser approach to correctness of asynchronous parallel programs is a flexible way to divide a correctness proof into several lemmas, no one of which requires both deep reasoning and explicit enumeration of all the control states required in the nondeterministic sequential form of the program. The approach is stated and justified abstractly, demonstrated in detail for a simple example program, and compared with other approaches to the correctness of parallel programs. The abstract formulation is independent of the model of parallelism in the example and can also be applied to nondeterminism not derived from asynchronous parallelism. We conclude with a survey of prospects for computer assisted proofs structured by the Church Rosser approach.

1. Introduction

Asynchronous parallel programs create situations where several processes might share some variables and be simultaneously ready to manipulate these variables in several ways. In general the processes do not notice each other and proceed at varying and unpredictable rates. The behavior of a program with asynchronous parallelism is therefore nondeterministic. At any moment, several actions are ready for performance, and we do not know which will actually be performed. As is usual in studies of digital computing, we assume that time is a succession of discrete instants, and the physical events that underline a computation take place between instants. As is usual in studies of asynchronous parallelism (e.g. [4, 5, 7, 10, 16, 23, 24]) we assume that actions are not simultaneously performed. From one instant to the next, only one of the actions ready to be performed will actually take place. In discussing *correctness* of programs we will only consider programs with inputs and outputs. A correct program always finishes its computation with outputs related in the desired way to the original inputs. Many

* A condensation of an earlier version of this paper was presented at the International Symposium on Proving and Improving Programs, Arc et Senans, France, July 1–3, 1975.

large programs, e.g. operating systems, have more complex correctness criteria. This paper is directly applicable only to portions of these programs that are expected to perform some definite service and then halt.

In general an asynchronous parallel program can be translated into sequential but nondeterministic form, with a distinct statement in the sequential program for each possible *set* of ready statements in the parallel one. For one programming formalism the translation is described in detail by Ashcroft and Manna [5]. One approach to proving correctness of a parallel program P is thus to replace it by a nondeterministic sequential program Q and apply the usual techniques [28] to Q. The combinatorial explosion that vitiates this approach is well known [4, 5, 17, 25].

Previous work in input/output correctness of parallel programs [5, 7] modifies the simple translation approach. From P we pass to a program P' which does whatever P does but does it with much less parallelism. Then we translate P' to a nondeterministic sequential program Q' and attempt to prove the correctness of Q'. The P to P' reprogramming methods in [5] are very complex and only defined for the particular low level formalism used there. The methods in [7] are simpler but even more restrictive in some ways. The key point is that P' must do A-and-then-B in most places where P does A-and-B-in-parallel. As will be seen in Section 6, such a translation is often inappropriate. Lipton [25] considers a reprogramming method wherein portions of a program P are enclosed in brackets to indicate that they are to be treated as instantaneous actions. Various properties other than input/output correctness are considered in [25], and under certain conditions a property of P can be verified by proving it for the less parallel program P'.

The Church–Rosser approach derives the correctness of P from the conjunction of four simpler properties to be established by four separate lemmas. Only the lemma for the "Church–Rosser property" will generally require explicitly enumerating the huge list of control states for the corresponding nondeterministic sequential program Q. Thanks to a theorem of Sethi [39] which reduces the Church–Rosser property to much simpler properties, the necessary reasoning is quite trivial for nearly all control states. (A theorem somewhat similar to Sethi's is stated by Nivat [34, Theorem 1].) In our example correctness proof, 164 cases will be reduced to two cases. The Church–Rosser approach can enhance reprogramming approaches [5, 7, 25] for input/output correctness. One difficulty with these approaches is the need to show that correctness of the new program P' implies correctness of the original program P. It will generally be much easier to show that "weak partial correctness" of P' implies the same property for P, and this is another of our four properties. We expect that the weak partial correctness lemma of a Church–Rosser correctness proof will often be demonstrated by reprogramming. The Church–Rosser and reprogramming approaches are complementary, not competitive.

The major limitations of the Church–Rosser approach in its present form should also be noted. The restriction to input/output correctness is inappropriate for many kinds of programs. The assumption of no strict simultaneity is probably unnecessary. By introducing at least one error value E and agreeing that the effect of *simultaneously* performing assignments $A \leftarrow 3$ and $A \leftarrow 5$ is to leave E in A, we could probably carry along the possibility of strict simultaneity at very modest cost.

Another limitation is the prohibition of **goto** in parallel programs. Since we can use **leave** as well as the classical structured programming control operators, the prohibition of **goto** is not in itself a very severe limitation. However, we also need to assume that the program P remains unchanged during all computations it generates. In concert with the lack of **goto,** this rules out recursive procedures. Cadiou and Levy [7] permit recursion. In principle recursion could be expressed by stacking and jumping in refinements of the low level formalisms of [4] and [5]; whether any program complex enough to require validation will remain intelligible enough to obtain it is another question.

The paper proceeds as follows. Background on relations, graphs, and structured programming is provided in Section 2. Theoretical discussions of structured programming control operators [21, 36] tend to emphasize the contrast with **goto**; we prefer to emphasize the common properties of these operators that facilitate correctness proofs. The informal discussions of such facilitation are too numerous and well known for review here, but formal discussions (as in [12] and the works cited there) have omitted **leave** and have been tied to other considerations. Parallelism affects Section 2 in only one way: we must use a set of flowchart nodes rather than a single node as the control state. Theorem 2.8 shows that the fundamental relation between cycles in flowcharts and infinite computations, though no longer obvious, is still true. In Section 3 we introduce asynchronous parallelism as an ordinary control operator like **if ... then ... else ...** and **begin ...; ...; ... end.** Like any other operator, the parallel compound statement **parbegin ... ; ... ; ... parend** needs to be defined. Of course, the definition is not the same as that used for **begin ... ; ... ; ... end,** but it does have the same form in the precise sense explained in Section 2. Parallelism is just added to whatever else is available without any unpleasant interactions. Section 3 also presents the program whose correctness we wish to prove. It is a high level version of an example used by Karp and Miller [16, Example 5.1]: we find the first positive component in an array of integers by searching even and odd indices in parallel. Each search can influence the other by changing the upper bound for indices in the other search. This program illustrates the advantages and dangers of parallelism.

Section 4 defines the finishing, weak partial correctness, admissibility, and Church–Rosser properties for any "abstract machine with a goal", including the abstract machine specified by a program that attempts to compute a relation between inputs and outputs. Theorem 4.6 states that these four properties together imply correctness. Readers interested in nondeterministic algorithms not derived from asynchronous parallelism may wish to begin with Section 4. The example correctness proof is presented in Section 5 and discussed in Section 6, from the viewpoint of providing computer assistance to humans in proofs too complex for computers or humans alone to carry out. Space does not permit more than one example, but remarks on the generality of the Church–Rosser approach appear in appropriate places. Within the limits already mentioned, the approach should be widely applicable. (See note added at end of paper.)

Several observations on program correctness proofs generally become especially apt when asynchronous parallelism is considered. One of these is not already well known and widely accepted, so we note it here while leaving all details to Section 6.

Analysis of control and data flow is needed anyway for optimization [2, 26]. Such analysis is *also* useful in correctness proofs and is often done informally by programmers. The role of formal flow analysis in proofs of "clean termination" has been systematically explored for sequential programs by Sites [40] (see also [35]).

The author thanks G. Markowsky, R. Sethi, and the referees for comments that led to several expository improvements and corrections.

2. Background

We use some rather common notation for relations and functions. If $\Rightarrow$ is a relation on a set S then $\Rightarrow^0$ is the equality relation on S and $\Rightarrow^k$ for any nonnegative integer k is defined inductively by $(\Rightarrow^{k+1}) = (\Rightarrow^k \Rightarrow)$. The union of the relations $\Rightarrow^k$ for all k is denoted $\Rightarrow^*$. More detailed explanations are available in many places, such as [37, Section 2, 39, Section 2].

Directed graphs will be used to analyze control flow in programs. Following the increasingly common algebraic style of definition, we specify a graph with sets N (of "nodes") and A (of "arcs") together with maps $s : A \to N$ and $t : A \to N$ called the "source" and "target" maps. An arc a runs from its source sa to its target ta. In our examples N and A will be finite and each arc a will be determined by the pair of nodes (sa, ta).

To deal with control flow in high level programs without obscuring them by translation into some low level language, we associate a *control flow graph* with each program. For example, consider the ALGOL 60 program

```
ANYBODY: begin integer INDEX;
  TOM: INDEX := 4;
  DICK: INDEX := INDEX + 7;
  HARRY: if INDEX < 39 then ASSIGN5: INDEX := 5
                       else ASSIGN8: INDEX := 8
end
```

whose control flow graph is shown in Fig. 2.1. This graph is larger than the equivalent low level flowchart shown in Fig. 2.2, but it is derived from the program text in a simpler way whose advantages become obvious when programs complex enough to be interesting are considered. With enough labelling and indentation, the control flow graph can easily be read off from the program text without the use of a separate picture. The derivation of a low level flowchart is much more complex [31, pp. 147–148].

In general, an occurrence of a simple statement leads to two nodes and one arc in the graph. If the statement has a label L, then the nodes are called *entering* L and *leaving* L. The actual execution of the statement is considered to take place when control flows along the arc from *entering* L to *leaving* L. This instantaneous view of simple statements is at variance with that of Dijkstra [10, p. 68], as will be discussed at the end of Section 3.

One of the major properties of high level languages is the use of *control operators* to construct complex statements from simpler statements as parts. The control

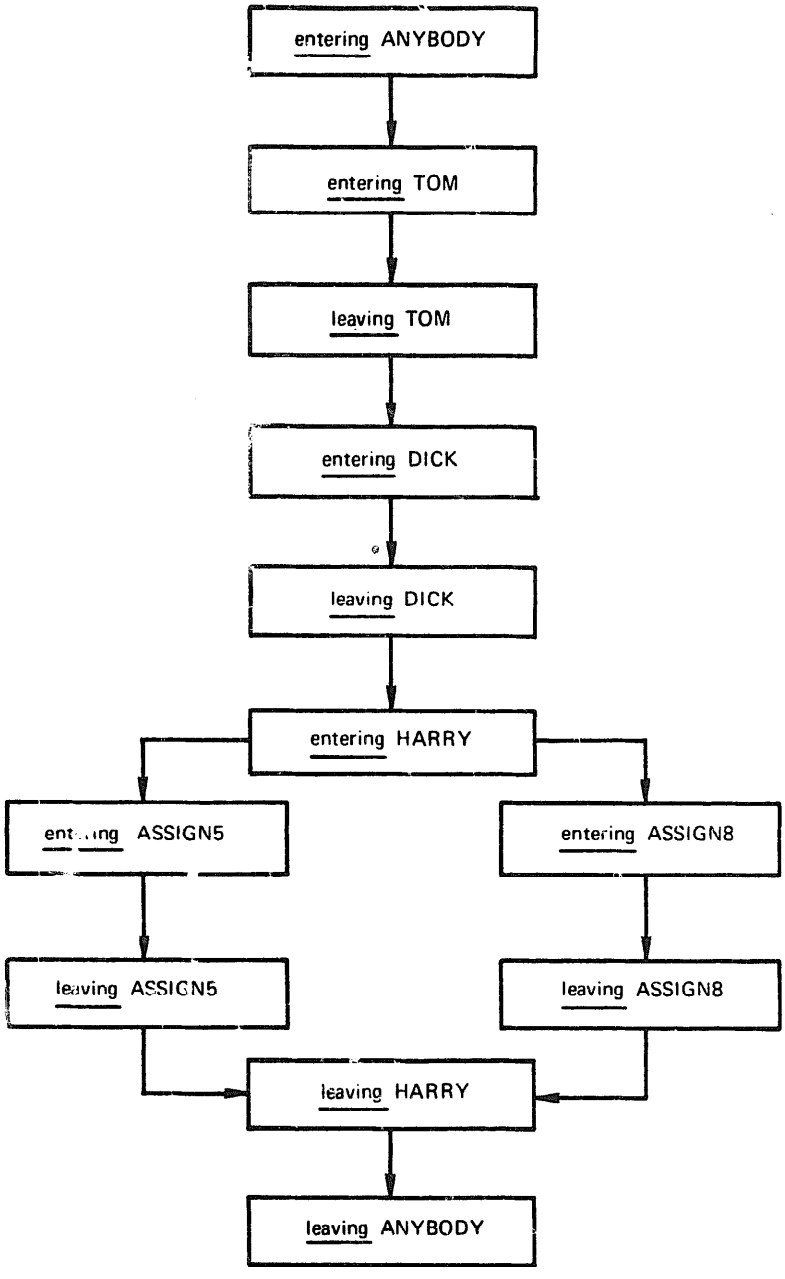

Fig. 2.1. Control flow graph for the program ANYBODY.

operators used in ANYBODY are the conditional operator, whose arguments are a Boolean expression and two statements, and the block operator, whose arguments are a list of declarations and a list of statements. In general, an occurrence of a complex statement labelled L will give rise to nodes *entering* L and *leaving* L, plus arcs joining these nodes to the nodes associated with statements in the argument list to the control operator. For example,

HARRY: **if**... **then** ASSIGN5: ... **else** ASSIGN8: ...

requires arcs from *entering* HARRY to *entering* ASSIGN5 and to *entering* ASSIGN8. Proper exit from the conditional requires arcs from *leaving* ASSIGN5 and from *leaving* ASSIGN8 to *leaving* HARRY. This simple bottom up structure

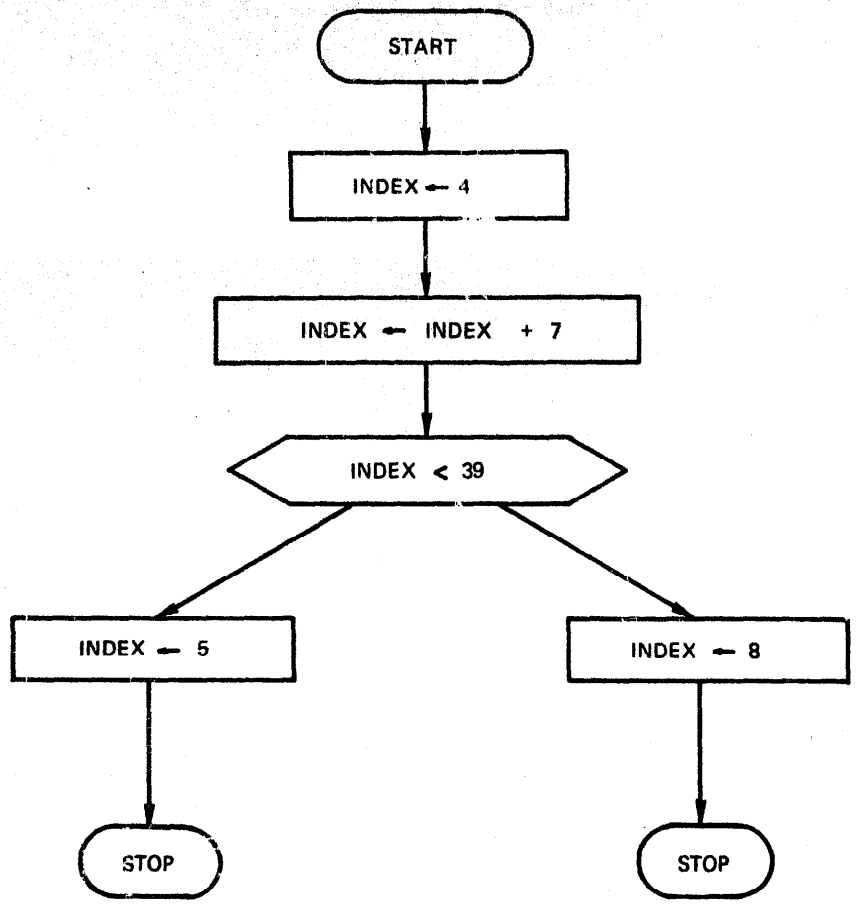

Fig. 2.2. Low level flow chart equivalent to the program ANYBODY.

tor the control flow graph is a characteristic of classical structured programming control operators: **if ... then ... else...**, **while ... do...**, **begin ...; ...; ...; ... end**, and so on. The mapping from program text to control flow graph is inductively specified by "initial algebra semantics" [12] or "synthesized attributes" [20].

Some control operators add more than two nodes to the graph, as in

ALPHA: **for** I := 1 **step** 2 **until** 83 **do** BETA:...

with new nodes

entering ALPHA (about to initialize *I*),
stepping ALPHA (about to increment *I*),
testing ALPHA (about to compare *I* with 83),
leaving ALPHA (about to do whatever is next).

This is no more difficult to work with than **if ... then ... else...**.

We have sketched a natural map from program text to control flow graph, assuming that the control operators are

if ... then ... else..., while ... do..., begin...; ...; ... end (2.1.1)

and so on, with straightforward Boolean expressions as the tests. In general, an ordinary structured programming control operator *C* may be used to form a (labelled) statement

$$\alpha\colon\ C(\beta_1, \cdots, \beta_K, \cdots) \tag{2.1.2}$$

in the program text, where the statements (with identifying labels) $\beta_1, \cdots, \beta_K$ are all the statement arguments to C. The set $N\alpha$ of nodes contributed by α to the control flow graph is

$$N\alpha = \bigcup_{k=1}^{K} N\beta_k \cup \{\textit{entering } \alpha, \textit{leaving } \alpha\} \cup S(C, \alpha), \tag{2.1.3}$$

where $S(C, \alpha)$ is a possibly empty set of nodes whose size depends only on C. For example, $S(C, \alpha)$ is $\{\textit{stepping } \alpha, \textit{testing } \alpha\}$ if C is the operator that forms stepped iteration statements. We assume there is a bound s_0 on the size of such sets:

$$|S(C, \alpha)| \leq s_0, \quad \text{for all } C. \tag{2.1.4}$$

The set $A\alpha$ of arcs contributed by α to the control flow graph has the form

$$A\alpha = \bigcup_{k=1}^{K} A\beta_k \cup T(C, \alpha), \tag{2.1.5}$$

where $T(C, \alpha)$ is a set of arcs whose members have sources and targets from among $\{\textit{entering } \beta_k \mid 1 \leq k \leq K\} \cup \{\textit{leaving } \beta_k \mid 1 \leq k \leq K\} \cup \{\textit{entering } \alpha, \textit{leaving } \alpha\} \cup S(C, \alpha)$. The size of $T(C, \alpha)$ depends only on C (and on K, if we allow for a varying length argument list). We assume there is a bound on the size of such sets:

$$|T(C, \alpha)| \leq 1 + t_0 K, \quad \text{for all } C \text{ and } K. \tag{2.1.6}$$

At least for sequential languages like ALGOL 60 the relation between the control flow graph and the meaning of the program should be clear enough intuitively that we can postpone a formal statement. The next item on our agenda is the exit operator **leave** ..., which is less well known than the classical operators summarized in (2.1) but is often indispensable for efficient structured programming. Our treatment of **leave** ... accords with [42, p. 413; 43, p. 68]; less intuitive formalizations of the same idea have been analyzed theoretically under the name "EXIT" [21, 36]. Other escape constructions can be treated similarly.

In a statement

$$\alpha\colon \textbf{leave } \beta \tag{2.2.1}$$

the argument β must be the identifying label of a statement which includes α. Here the label β is obligatory, not just a convenient way to name a portion of the program text. The nodes contributed by α to the control flow graph are just

$$N\alpha = \{\textit{entering } \alpha, \textit{leaving } \alpha\}, \tag{2.2.2}$$

but the arc contributed is

$$(\text{from } \textit{entering } \alpha \text{ to } \textit{leaving } \beta). \tag{2.2.3}$$

Using "inherited attributes" [20] to deal with **leave**, we have a simple map from program texts to control flow graphs. The control flow graph is finite and its size is bounded in terms of a natural measure of the size of the program text.

Lemma 2.3. *Let P be a program and let N (resp. A) be the set of nodes (resp. arcs) in the control flow graph of P. Let $\|P\|$ be the total number of statements in P. Then some $s_0 \geq 0$ has $|N| \leq (2+s_0)\|P\|$. Moreover, some $t_0 \geq 0$ has $|A| \leq (1+t_0)\|P\| - t_0$.*

Proof. In (2.2.2) a statement α: **leave** β contributes two nodes to N. In (2.1.3) and (2.1.4), a statement $\alpha: C(\beta_1, \cdots, \beta_K, \cdots)$ contributes

$$|N\alpha| \leq 2 + s_0 + \sum_{k=1}^{K} |N_{\beta_k}|$$

nodes to N. Summing over all α leads to $(2+s_0)\|P\|$. Similarly for the other inequality. □

We consider the meaning of a program to be a relation from values for designated *input* variables to values for designated *output* variables. Such variables could in practice be whole files, not just what language reference manuals call "variables". For our purposes it will be convenient to define such a relation in terms of the actions of an abstract interpreter. The *state* ξ of the interpreter is a map from variables to values. In addition to the variables mentioned in the program, there are special variables PROG and CONTROL. The value ξPROG is the program itself, which may perhaps have been modified in the course of the computation. The value ξCONTROL is called a *control state* and must be a set of nodes in the control flow graph of ξPROG.

Sequential languages could let control states be nodes because ξCONTROL is always $\{n\}$ for a single node n. There is only one thing ready to be done next at any time. In this case our model will reduce to the obvious definitions of how control flows through ξPROG in the ALGOL 60 interpreter implicit in [32], plus the obvious definition of **leave**. In general, however, our model will allow for parallelism without perturbing the meanings of ordinary control operators. With parallelism there may be several nodes in ξCONTROL because several processes are simultaneously active.

For any state ξ the language definition determines a (possibly empty) set of states η that could be reached in one step from ξ. We write this *transition relation* as $\xi \Rightarrow \eta$. For example, the program ANYBODY includes the statement

$$\text{DICK:INDEX} := \text{INDEX} + 7. \tag{2.4.1}$$

Any state ξ with *entering* DICK in ξCONTROL is ready to add 7 to INDEX, so $\xi \Rightarrow \eta$ with

$$\begin{aligned} &\eta\text{CONTROL} = (\xi\text{CONTROL} - \{\textit{entering}\ \text{DICK}\}) \cup \{\textit{leaving}\ \text{DICK}\}; \\ &\eta\text{INDEX} = \xi\text{INDEX} + 7; \\ &\eta \text{ like } \xi \text{ otherwise.} \end{aligned} \tag{2.4.2}$$

During this transition we say that control *follows* the arc from *entering* DICK to *leaving* DICK. The effects of other simple statements (other than procedure calls) may be defined similarly. We assume that the effects of any procedure calls have

already been determined, so that calls can be treated like ordinary simple statements.

Now consider any of the classical control operators C from (2.1). The conditional is typical. For

HARRY: **if** INDEX < 39 **then** ASSIGN5: ... **else** ASSIGN8: ... (2.5.1)

we have two rules. First, any state ξ with *entering* HARRY in ξCONTROL has $\xi \Rightarrow \eta$ with

ηCONTROL = (ξCONTROL − {*entering* HARRY}) ∪ {*entering* β}

where β is ASSIGN5 if ξINDEX < 39 and is ASSIGN8 otherwise;

η like ξ otherwise. (2.5.2)

Second, consider any state ξ with *leaving* β in ξCONTROL, where β is ASSIGN5 or ASSIGN8. Then $\xi \Rightarrow \eta$ with

ηCONTROL = (ηCONTROL − {*leaving* β}) ∪ {*leaving* HARRY};

η like ξ otherwise. (2.5.3)

In both cases, (2.5.2) and (2.5.3), we have the obvious definition of the arcs *followed* by the transition $\xi \Rightarrow \eta$.

Finally, consider the unusual control operator,

α: **leave** β. (2.6.1)

Any state ξ with *entering* α in ξCONTROL may have other nodes of $N\beta$ in ξCONTROL as well. We remove *all* of them in $\xi \Rightarrow \eta$ with

ηCONTROL = (ξCONTROL − $N\beta$) ∪ {*leaving* β};

η like ξ otherwise. (2.6.2)

The arc from *entering* α to *leaving* β is the one *followed* here, and ηCONTROL contains only *leaving* β from $N\beta$. Summarizing some useful consequences of (2.4)–(2.6) and similar definitions, we note the following lemma.

Lemma 2.7. *For each transition $\xi \Rightarrow \eta$ such that ξPROG = ηPROG there is a nonempty set $AF(\xi, \eta)$ of arcs followed in the transition, where each a in $AF(\xi, \eta)$ is an arc in the control flow graph of ξPROG. The control states are related by*

$$\eta\text{CONTROL} = (\xi\text{CONTROL} - Q) \cup \{ta \mid a \text{ in } AF(\xi, \eta)\}, \quad (1)$$

where Q is subset of ξCONTROL such that

$$\{sa \mid a \text{ in } AF(\xi, \eta)\} \subset Q. \quad (2)$$

But for **leave**, the inclusion in (2) above could be an equation. The possibility that $|\xi\text{CONTROL}| > 1$ renders the following result in need of proof.

Theorem 2.8. *Let $\xi_0 \Rightarrow \xi_1 \Rightarrow \xi_2 \cdots$ be an infinite computation (for an initial state ξ_0) by program P such that ξ_i PROG = P for all i. Let B be any set of arcs in the control flow graph such that removal of B would leave an acyclic graph. There is an arc b in B such that control follows b infinitely often.*

Proof. We seek to get b in $AF(\xi_i, \xi_{i+1})$, for infinitely many i in Lemma 2.7. By Lemma 2.3 it will suffice to construct an infinite sequence $(a_0, a_1, a_2, \cdots)$ of arcs such that, for all i,

$$a_i \text{ is in } AF(\xi_i, \xi_{i+1}) \text{ and } ta_i = sa_{i+1}. \tag{1}$$

We begin by constructing a sequence $(T_0, T_1, \cdots)$ of finite rooted trees. Each tree node is marked by a node in the control flow graph. Each tree arc (from a tree node to one of its children) is marked by an arc in the control flow graph.

Let T_0 have just a root, marked by the entry node of the control flow graph. Thus for $i = 0$ we have

$$\xi_i \text{CONTROL} \subset \{n \text{ in } N \mid n \text{ marks a leaf in } T_i\}. \tag{2}$$

Given T_i such that (2) holds, we form T_{i+1} by adding children to some of the leaves of T_i. For each leaf λ in T_i, let n be the node marking λ. For each a in $AF(\xi_i, \xi_{i+1})$ such that $sa = n$, we give λ a child marked by ta and we mark the new tree arc by a.

By Lemma 2.7, T_{i+1} satisfies (2) also, and the contruction continues inductively. Moreover, T_i is properly included in T_{i+1} for all i. The sequence of finite trees converges to an infinite tree T_∞ in which each node has finitely many children. König's lemma supplies an infinite path in T_∞, and the sequence of control flow arcs marking tree arcs along such a path satisfies (1). □

The fixed program restriction ξ_iPROG = P in the above theorem is not so innocuous as it looks. Without **goto** we can only interpret procedure calls by macroexpansion, as in the ALGOL 60 copy rule [32]. In saying that PROG does not vary we are saying that all necessary macroexpansions could be performed *before* computation begins. This is not true for many programs with recursive procedures. The question whether it is true for a given program is undecidable [22, Section 4]. The parts of our work that use Theorem 2.8 are thus restricted to nonrecursive programs. For these programs it is a powerful tool for termination proofs. Some tools for termination proofs applicable to recursive parallel programs are provided by Cadiou and Levy [7]. (See note added at end of paper.)

3. Asynchronous parallel programming

Dijkstra [9, Section 2, 10, Section 2.1] adds a parallel compound statement construction to ALGOL 60 and similar sequential languages. In [9] the effect of α: **parbegin** β: ...; γ: ... **parend** is considered to be almost self-explanatory: just do β and γ "in parallel". There is more to be said, as the lengthy discussion in [10] indicates. Can the parallel compound statement be introduced in a precise way as an ordinary structured programming control operator? Can

parbegin ...; ...; ... **parend** (3.1.1)

join the classical operators listed in (2.1.1)? Indeed it can. Consider a labelled statement

α: **parbegin** β_1; ...; β_K **parend** (3.1.2)

in the program text. The set $N\alpha$ of nodes contributed to the control flow graph by α is

$$N\alpha = \bigcup_{k=1}^{K} N\beta_k \cup \{\textit{entering } \alpha, \textit{leaving } \alpha\}. \tag{3.1.3}$$

The set $A\alpha$ of arcs contributed to the control flowgraph is:

$$A\alpha = \bigcup_{k=1}^{K} A\beta_k \cup \{b_k \mid 1 \leq k \leq K\} \cup \{c_k \mid 1 \leq k \leq K\}, \tag{3.1.4}$$

where each b_k runs from *entering* α to *entering* β_k and each c_k runs from *leaving* β_k to *leaving* α.

The effect (3.1) of the parallel compound statement is a special case of the effect (2.1) of an ordinary control operator. Thus Lemma 2.3 is unaffected by parallelism.

The transition rule for parallel compound statements is straightforward. Consider

α: **parbegin** β_1; ...; β_K **parend.** (3.2.1)

Any interpreter state ξ with *entering* α in ξCONTROL is ready to begin all K statements in parallel, so $\xi \Rightarrow \eta$ with

$$\eta\text{CONTROL} = (\xi\text{CONTROL} - \{\textit{entering } \alpha\}) \cup \{\textit{entering } \beta_k \mid 1 \leq k \leq K\};$$

$$\eta \text{ like } \xi \text{ otherwise.} \tag{3.2.2}$$

Any state ξ with all *leaving* β_k in ξCONTROL is ready to leave α, so $\xi \Rightarrow \eta$ with

$$\eta\text{CONTROL} = (\zeta\text{CONTROL} - \{\textit{leaving } \beta_k \mid 1 \leq k \leq K\}) \cup \{\textit{leaving } \alpha\};$$

$$\eta \text{ like } \xi \text{ otherwise.} \tag{3.2.3}$$

Note the similarity to the rule (2.5) for the conditional. Lemma 2.7 and Theorem 2.8 are unaffected by parallelism. In particular, the size of the control flow graph is linear in the number of program statements even though the number of control states is not! The program whose correctness we will prove illustrates this observation. Although very simple, it clearly displays the peculiarities of asynchronous parallel activity by processes which share some of their variables. The program FINDPOS is much like an example used by Karp and Miller [16, Example 5.1]. Given an array x of integers, we wish to find the first positive component $x[k]$, if any. There are 164 control states but only 44 nodes in the control flow graph.

```
FINDPOS: begin integer array X; integer M, Z;
  comment indices of X run from 1 to M;
  comment X, M are inputs while Z is output;
```

```
integer I, J, K, EVENTOP, ODDTOP;
INITIALIZE: I := 2; J := 1;
EVENTOP := M + 1; ODDTOP := M + 1;
SEARCH: parbegin
            EVENSEARCH: while I < ODDTOP do
                EVENTEST:
                if X[I] > 0
                    then EVENYES:
                        begin FINDEVEN: EVENTOP := I;
                            leave EVENSEARCH
                        end
                    else EVENNO: I := I + 2;
            ODDSEARCH: while J < EVENTOP do
                ODDTEST:
                if X[J] > 0
                    then ODDYES:
                        begin FINDODD: ODDTOP := J;
                            leave ODDSEARCH
                        end
                    else ODDNO: J := J + 2
parend;
SETK: K := min(EVENTOP, ODDTOP);
FINISH: if K > M then Z := 0 else Z := X[K]
end
```

Before stating the program we said informally what it tried to do. Now it is time to be more precise. The *initial* state ξ of the interpreter for inputs x and m has

$$\xi X = x \text{ and } \xi M = m;$$

$$\xi\text{PROG} = \text{FINDPOS and } \xi\text{CONTROL} = \{\textit{entering } \text{FINDPOS}\};$$

$$\text{all other variables defaulted.} \tag{3.3.1}$$

A *final* state η of the interpreter has

$$\eta\text{PROG} = \text{FINDPOS and } \eta\text{CONTROL} = \{\textit{leaving } \text{FINDPOS}\}. \tag{3.3.2}$$

Only ηZ is of interest to us in a final state. The desired relation between inputs and outputs is expressed by the *goal function* G which maps pairs (initial state, final state) into $\{0, 1\}$: $G(\xi, \eta) = 1$ iff

$$\begin{aligned} &(\eta Z = 0 \text{ and } \xi X[h] \leqslant 0, \text{ all } 1 \leqslant h \leqslant \xi M) \text{ or} \\ &(\eta Z = \xi X[k] \text{ for } k = min\{h \mid \xi X[h] > 0\}). \end{aligned} \tag{3.3.3}$$

Simple though it is, FINDPOS illustrates the advantages of parallelism. The processes EVENSEARCH and ODDSEARCH running in parallel are roughly twice as fast as a single search through the indices. The variables EVENTOP and ODDTOP are shared by the processes in a special way: a variable used in one

process may be changed by another. The dangers in this situation are clear. But without such sharing it is difficult to see how EVENSEARCH can tell ODDSEARCH not to bother searching past $X[14]$. By sharing a *few* variables in this way, parallel processes can communicate important matters without becoming utterly mystifying to human intuition. Due caution must be exercised in correctness proofs, however. Although it correctly computes a singlevalued function, FINDPOS is *not* determinate [16, Definition 1.9]. Before defining the abstractions that do lead to a correctness proof, we prove as much as possible now.

Lemma 3.4. *For any array $x[1:m]$ of integers, all computations by* FINDPOS *with inputs x to X and m to M are finite.*

Proof. Let B be the set of arcs in the control flow graph consisting of the arc from *entering* EVENNO to *leaving* EVENNO and the arc from *entering* ODDNO to *leaving* ODDNO. Then I or J increases by 2 whenever control follows an arc in B. When other arcs are followed, I and J are not changed. By Theorem 2.8, the set of values assumed by I or by J during an infinite computation must be unbounded. To show that all computations are finite, it will suffice to show that, for some bounds b, c, the assertion

$$\xi I \leq b \text{ and } \xi J \leq c \tag{1}$$

is true for each state ξ in a computation such that ξCONTROL includes *leaving* EVENNO or *leaving* ODDNO. To prove (1) for the needed states ξ we will consider a stronger assertion and prove that it becomes true and remains true in any computation where control reaches *leaving* EVENNO or *leaving* ODDNO. The assertion to be used is

$$I \leq M+2 \text{ and } J \leq M+2 \text{ and ODDTOP} \leq M+1 \text{ and EVENTOP} \leq M+1. \tag{2}$$

Now (2) is true for any ξ with ξCONTROL = {*entering* SEARCH}, and such a state must precede the states of interest to us in any computation. A transition $\xi \Rightarrow \eta$ will preserve (2) trivially if it changes none of the variables mentioned in (2). The only transitions that do change these variables follow the arcs

$$\text{(from } \textit{entering} \text{ FINDEVEN to } \textit{leaving} \text{ FINDEVEN)}, \tag{3}$$

$$\text{(from } \textit{entering} \text{ EVENNO to } \textit{leaving} \text{ EVENNO)}, \tag{4}$$

and the similar arcs within ODDSEARCH.

For (3) we have $\xi \Rightarrow \eta$ with arc (3) in $AF(\xi, \eta)$ and (2) true for ξ and all previous states. Therefore,

$$\eta I = \xi I \leq m+2 \text{ and } \eta J = \xi J \leq m+2;$$

$$\eta\text{ODDTOP} = \xi\text{ODDTOP} \leq m+1;$$

$$\eta\text{EVENTOP} = \xi I.$$

We claim $\xi I \leq m+1$. Since *entering* EVENSEARCH lies on every control path from *entering* FINDPOS to *entering* FINDEVEN, we know that ξ is preceded by a

state ζ such that ζCONTROL contains *entering* EVENSEARCH and $\zeta I < \zeta$ODDTOP. Therefore $\zeta I \leq m + 1$ by the induction hypothesis. Since I cannot be updated outside of EVENSEARCH, the most recent such ζ has $\xi I = \zeta I$ and hence $\xi I \leq m + 1$.

For (4) we have $\xi \Rightarrow \eta$ with arc (4) in $AF(\xi, \eta)$ and (2) true for ξ and all previous states. Therefore

$$\eta I = \xi I + 2 \text{ and } \eta J = \xi J \leq m + 2;$$

$$\eta\text{ODDTOP} = \xi\text{ODDTOP} \leq m + 1;$$

$$\eta\text{EVENTOP} = \xi\text{EVENTOP} \leq m + 1.$$

We claim $\xi I \leq m$. Since I cannot be updated outside of EVENSEARCH, $\xi I = \zeta I$ for the most recent ζ such that ζCONTROL contains *entering* EVENSEARCH. By the induction hypothesis,

$$\zeta I < \zeta\text{ODDTOP} \leq m + 1, \text{ so that } \zeta I \leq m \text{ and } \xi I \leq m.$$

Similar reasoning applies to the arcs like (3) and (4) but within ODDSEARCH. □

An inductive proof that an assertion is invariant (i.e. becomes and remains true) is of course the fundamental concept in program correctness proving [11, 33]. Even simple assertions like (2) in the above lemma can be difficult. In a sequential program we could conclude that $\xi I = \zeta I$ when control passes from *entering* EVENSEARCH in ζ to *entering* FINDEVEN in ξ; I is not changed along the way. Parallelism raises the possibility that I may be changed anyway by statements not on a direct path from *entering EVENSEARCH to entering* FINDEVEN. Fortunately, no such statements exist outside of EVENSEARCH. (The statement initializing I is not relevant this late in a computation.) In effect our inductive invariance proof could stay within EVENSEARCH and ignore whatever other processes might be active. Recognition of such situations is discussed at length by Lauer [23, pp. 58–74] (see also Ashcroft [4] and Cadiou and Levy [7]).

Correctness is more than the finiteness of computations. Any computation starting at an initial state ξ should reach a final state η such that $G(\xi, \eta) = 1$, where G is the goal function from (3.3). The next section provides the additional mathematical machinery needed to complete the correctness proof. This section concludes with remarks that should be helpful at this point to readers who are uneasy about our failure to discuss synchronization, the finite delay property, and similarly subtle aspects of parallelism. Others may wish to go directly to Section 4.

Synchronization is the use of special global variables and operations which let parallel processes cooperate despite a generally asynchronous mode of operation. Lipton [24] analyzes the relative capabilities of various "synchronization primitives" that have been proposed for such purposes, and further references may be found in [24]. The Church–Rosser approach is independent of the choice of synchronization primitives. It happens that the example program FINDPOS uses none. The general mathematical formulation never assumes that synchronization is present *or* absent.

The intuition in the finite delay property is easily stated: any action ready to be performed will eventually be performed. This property is difficult to formalize [16, Definition 1.6 (iii)] but is sometimes crucial in correctness proofs. It is easy to write a program which has some infinite computations from our point of view, but whose infinite computations all involve infinite delays. Such a program will always terminate but we will be unable to prove it. On the other hand, a termination proof that does not use finite delay would only be obscured by mention of this difficult concept. The abstract formulation of the Church–Rosser approach in the next section is independent of finite delay.

An unusual and slightly oversimplified aspect of our view of parallelism is the instantaneous execution of simple statements. Statements which do not syntactically include other statements may still involve many machine instructions on most real computers. Instantaneity here is at variance with Dijkstra's informal semantics of $S := S + 1$ when S is a variable common to two processes [10, p. 68]. However, a slightly earlier discussion of a single sequential process considers execution of $j := j + 1$ to be instantaneous [10, pp. 47–52 (esp. Remark 3)], and we prefer to introduce parallelism without radical departures from what is appropriate in sequential computing.

To put the question in perspective, consider several language levels. The author or certifier [6] of a microprogram pretends that microinstructions are executed instantaneously. This is not physically true but it may be assumed *without loss of generality*. When microprogramming is complete, the (sequential) machine language programmer assumes without loss of generality that machine instructions are instantaneous. In the low or medium level languages commonly considered in mathematical studies of parallelism [e.g. 4, 5, 16, 23, 24, 25], instantaneity is also assumed. None of these works consider what may happen *while* an elementary action like P (semaphore) or V (semaphore) is being performed. The only thing unusual here is that we have consistently applied this assumption to high level languages such as ALGOL 60. Under the noninstantaneous view from [10], the introduction of parallelism requires either a sudden explosion of detail throughout a language definition or a sudden drastic loss of precision. In executing $S := S + 1$, is the fetching of S for the addition considered instantaneous? Is it possible that we fetch a few bytes of S, then some other process changes S, and then we fetch the rest of S? Without precise answers to such questions we can hardly discuss program reliability when parallel processes can share variables. The noninstantaneous view forces additional machine dependence upon us. To perform $S := S + 1$ we fetch S into a register, add 1 to what is in the register, and then store the register's contents into S. All of which would be quite silly if S happened to be stored in a portion of memory with substantial distributed processing capability and if there happened to be a machine instruction to the effect of "add one to yourself". Instantaneity avoids an awkward mixing of high level language semantics with details of machine architecture.

At any language level we need locking and tie breaking facilities [13] in order to assume instantaneity without loss of generality and with reasonable efficiency. An "instantaneous" action can lock all of the variables involved, so that other "instantaneous" actions cannot use these variables until the first action has finished.

"Instantaneous" actions that share no variables *can* overlap in physical time, and this is an important advantage of locking [13] over critical sections [10, p. 53] in some situations. In a well structured language it should be easy to ascertain which variables are involved in more than one process, and only for those variables will a compiler that guarantees high level instantaneity need to put lock setting and lock testing operations into the code generated for a simple statement.

The oversimplification in high level instantaneity arises from the doublethink associated with certain kinds of simple statements in a high level language. A procedure call is usually a simple statement, but its effect is understood to be that of a possibly very complicated compound statement. To guarantee high level instantaneity in a procedural language, we would probably have to lock a great many variables for each procedure call. It is more natural here to keep a procedure call strictly equivalent to the corresponding compound statement. A statement whose definition has the form "the effect is as if the programmer had written..." may be syntactically simple, but the semantics come from the expansion to an equivalent and syntactically complex statement in the source language. Properly qualified, high level instantaneity applies only to statements not expanded in the source language, regardless of the length or form of the discussion in the language reference manual which somehow explains them.

4. Abstract machines with goals

The class of mathematical entities considered to be "machines" here is broad enough to include defining interpreters for low level programs in microcode or assembly languages as well as high level programs in procedural or nonprocedural languages. In particular, the abstract machine defined by our example program FINDPOS is included, with interpreter states serving as "states" in the following definition. Unlike [6, 30], we allow nondeterminism.

Definition 4.1. A *machine* is a quintuple $M = (S, \Rightarrow, S_i, S_f, G)$ where S is a set whose members are called *states*, $\Rightarrow$ is a relation on S called the *transition relation*, S_i and S_f are subsets of S called the *initial* and *final* states, and G is a map from $S_i \times S_f$ into $\{0, 1\}$ called the *goal function*. We further require that each final state η be a *dead* state: no ζ in S has $\eta \Rightarrow \zeta$.

Quite apart from its nondeterminism, our definition is unusual in two important respects. First, we do not identify dead states with final states. Thus we leave room for nonfinal dead states to model the common practical situation where a program stops because of an error condition rather than because it has finished its work. Second, we include G in M: one cannot specify a machine without specifying what it is supposed to do.

Definition 4.2. Let ξ be a state of the machine M. A *computation by M for* ξ is any maximal sequence $(\xi_0, \xi_1, ...)$ of states such that $\xi_0 = \xi$ and $\xi_h \Rightarrow \xi_{h+1}$ for all h. A *finishing* computation is one which is finite and ends with a final state.

Definition 4.3. Let ξ be an initial state of the machine M. Then M is *correct for* ξ if every computation C by M for ξ finishes with a final state η (which may depend on C) such that $G(\xi, \eta) = 1$. If M is correct for each initial state then M is *correct.*

To finish is to "terminate cleanly" [40]. Scholars enamored with latinate vocabulary might prefer "terminate normally", but "finish" is really quite adequate. Correctness can be demonstrated by separate proofs that all computations finish and that all finishing computations satisfy the goal function. This last property is *partial correctness* [29]. Nondeterminism by itself does not discredit the usual divide-and-conquer strategy that separates finishing from partial correctness. Determinism is assumed in most of the many formal discussions of inductive invariance proofs [27] but is not essential [28]. Unfortunately, the assertion whose truth is invariant in a partial correctness proof has a general form [27] which branches according to the value of CONTROL when ξ is an interpreter state. Parallelism explodes the number of control states and makes the invariant assertion for a partial correctness proof unmanageably large. Our approach to this difficulty is to replace partial correctness by a weaker property.

Definition 4.4. Let ξ be an initial state for a machine M. Then M is *finishing for* ξ iff every computation by M for ξ finishes. On the other hand, M is *weakly partially correct for* ξ iff there is a computation C by M for ξ such that, if C finishes with a final state η, then $G(\xi, \eta) = 1$.

To parlay finishing and weak partial correctness into correctness, we want to show that no computation can be very different from the C of the definition. Consideration of things that are different but alike enough for some purposes leads naturally to equivalence relations.

Definition 4.5. Let S be the set of states for the machine M and let $\sim$ be an equivalence relation on M. The set of *reachable* states is

$$S_r = \{\eta \text{ in } S \mid \xi \Rightarrow^* \eta, \text{ some } \xi \text{ in } S_i\}. \tag{1}$$

The *replacement system* for M and $\sim$ is the triple

$$\text{Rep}(M, \sim) = (S_r, \Rightarrow_r, \sim_r), \tag{2}$$

where $\Rightarrow_r$ (resp. $\sim_r$) is $\Rightarrow$ (resp. $\sim$) restricted to the reachable states. The replacement system is *admissible* iff all final states η, ζ in S_r have

$$\eta \sim \zeta \text{ implies } (G(\xi, \eta) = G(\xi, \zeta), \text{ all } \xi \text{ in } S_i). \tag{3}$$

The replacement system is *Church–Rosser* iff, for all ξ, ξ', η, η' in S_r such that

$$\xi \Rightarrow^* \eta \text{ and } \xi' \Rightarrow^* \eta' \text{ and } \xi \sim \xi', \tag{4}$$

there are ζ, ζ' in S_r such that

$$\eta \Rightarrow^* \zeta \text{ and } \eta' \Rightarrow^* \zeta' \text{ and } \zeta \sim \zeta'. \tag{5}$$

Replacement systems and the Church–Rosser property have several other

applications in computer science [1, 14, 37, 39]. A corollary of Theorem 3.5 and Lemma 3.6 in [37] has been applied to several problems involving parallelism [17] (see also [8]). Admissibility is relative to M, but the Church–Rosser property belongs to $(S_r, \Rightarrow_r, \sim_r)$ itself.

Theorem 4.6. *Let M be a machine and let $\sim$ be an equivalence relation among the states of M. Suppose that M is finishing and weakly partially correct while* Rep$(M, \sim)$ *is admissible and Church–Rosser. Then M is correct.*

Proof. Let ξ be any initial state of M, and let C be any computation by M from ξ. By the finishing property, C ends with a final state η. We must show $G(\xi, \eta) = 1$. By weak partial correctness there is also a computation C' ending with a final state η' such that $G(\xi, \eta') = 1$. Since $\xi \Rightarrow^* \eta$ and $\xi \Rightarrow^* \eta'$, the Church–Rosser property supplies us with states ζ, ζ' such that $\eta \Rightarrow^* \zeta$ and $\eta' \Rightarrow^* \zeta'$ and $\zeta \sim \zeta'$. But η and η' are dead states, so $\eta = \zeta$ and $\eta' = \zeta'$. Thus $\eta \sim \eta'$. By admissibility, $G(\xi, \eta) = G(\xi, \eta') = 1$. □

Thus correctness can be derived from four lemmas, two of which involve $\sim$ as well as M. We are free to choose $\sim$ in any way we like. The problem of choosing $\sim$ wisely is left open here. How might we prove a lemma to the effect that Rep$(M, \sim)$ is Church–Rosser? Sethi [39] has a result which is crucial to our approach. Theorem 2.2 of [39] relates the Church–Rosser property to weaker properties. By applying this theorem and König's lemma to abstract machines, we can derive the following lemma. (Explanations of "P1" and "P3" below appear in the proof of Lemma 5.5.)

Lemma 4.7. *Let M be a machine and let $\sim$ be an equivalence relation on the set S of states of M. Suppose M is finishing and suppose each ξ in S has $\xi \Rightarrow \eta$ for only finitely many η in S. Then* Rep$(M, \sim)$ *is Church–Rosser iff it is both* P1 *and* P3.

5. Example correctness proof

The program FINDPOS presented in Section 3 will now be proved correct with the aid of the tools explained in Section 4. The proof is structured by Theorem 4.6 and is very detailed in those portions that illustrate unusual features of our approach or unusual difficulties associated with parallelism. Other portions are presented more briefly. In the next section we will discuss prospects for computer assisted proofs.

Lemma 5.1. *For any array $x[1:m]$ of integers,* FINDPOS *finishes all computations with inputs x to X and m to M.*

Proof. By Lemma 3.4, all computations are finite. All reachable dead states are final states, so all computations end in final states. □

Lemma 5.2. *For any array $x[1:m]$ of integers,* FINDPOS *is weakly partially correct with inputs x to X and m to M.*

Proof. We specify a computation $\xi_0 \Rightarrow \xi_1 \Rightarrow \xi_2 \Rightarrow \ldots$ by FINDPOS for the initial state ξ_0 determined by x and m as in (3.3.1). In this one computation the sequence of events is that EVENSEARCH finishes before ODDSEARCH even starts. If ξ_hCONTROL contains a node in the control flow graph (other than *leaving* EVENSEARCH) derived from EVENSEARCH, then the arcs followed in $\xi_h \Rightarrow \xi_{h+1}$ are all derived from EVENSEARCH. This computation finds the first even index (if any) for a positive x component and changes EVENTOP accordingly. Then it finds the first odd index (if any) for a positive x component that is below the previously established EVENTOP value. After leaving SEARCH, FINDPOS executes SETK and FINISH to reach a final state η with $G(\xi_0, \eta) = 1$ in (3.3.3). □

Now we must define an equivalence relation $\sim$ among interpreter states. For most control states Q we agree that any η with ηCONTROL $= Q$ has $\eta \sim \eta'$ iff $\eta = \eta'$. The only control states we must individually mention in defining $\sim$ are the exceptions. The first exception is the control state $Q_f = \{$*leaving* FINDPOS$\}$ involved in final states of the abstract machine. If ηCONTROL is Q_f then only the output variables matter:

$$\eta \sim \eta' \text{ iff } \eta\text{CONTROL} = \eta'\text{CONTROL and } \eta Z = \eta' Z. \tag{5.3.1}$$

Reasoning backward from Q_f to other single node control states, we define $\sim$ in terms of equality of control states and equality of variables (or functions of variables) that might appear in the output. For example, if ηCONTROL is $\{$*entering* FINISH$\}$, then

$$\begin{aligned} &\eta \sim \eta' \text{ iff } \eta\text{CONTROL} = \eta'\text{CONTROL and} \\ &\eta K = \eta' K \text{ and } \eta M = \eta' M \text{ and } \eta X = \eta' X. \end{aligned} \tag{5.3.2}$$

On the other hand, if ηCONTROL is $\{$*entering* SETK$\}$, then

$$\begin{aligned} &\eta \sim \eta' \text{ iff } \eta\text{CONTROL} = \eta'\text{CONTROL and} \\ &\min(\eta\text{EVENTOP}, \eta\text{ODDTOP}) = \min(\eta'\text{EVENTOP}, \eta'\text{ODDTOP}) \\ &\quad \text{and } \eta M = \eta' M \text{ and } \eta X = \eta' X. \end{aligned} \tag{5.3.3}$$

The last control state considered is $\{$*leaving* SEARCH$\}$, whose only predecessor states have more than one node in them. If ηCONTROL is this state then (5.3.3) is again used. No more special cases in the definition of $\sim$ will be needed. Using the abstract machine specified by FINDPOS and the equivalence relation $\sim$ just defined, we get the replacement system Rep(FINDPOS, $\sim$) in Definition 4.5. Since the goal function G only involves the output variable in the final state, (5.3.1) implies the next lemma immediately.

Lemma 5.4. *The replacement system* Rep(FINDPOS, $\sim$) *is admissible for* FINDPOS.

Lemma 5.5. *The replacement system* Rep(FINDPOS, $\sim$) *is Church–Rosser.*

Proof. By Lemmas 4.7 and 5.1, it will suffice to verify the properties P1 and P3 described below.

Consider any reachable states ξ, ξ', η with $\xi \sim \xi'$ and $\xi \Rightarrow \eta$. For P1 we must show that states ζ, ζ' have $\eta \Rightarrow^* \zeta$ and $\xi' \Rightarrow^* \zeta'$ and $\zeta \sim \zeta'$. If $\xi = \xi'$ we may let ζ and ζ' be η. Now assume $\xi \neq \xi'$, so that $\xi \sim \xi'$ because of one of the special cases in the definition of $\sim$. In all cases we may let ζ be η and ζ' be the unique state with $\xi' \Rightarrow \zeta'$. The definition was deliberately rigged to prevent $\xi \sim \xi'$ unless $\zeta \sim \zeta'$. This proves P1.

Consider any reachable states ξ, η, η' with $\xi \Rightarrow \eta$ and $\xi \Rightarrow \eta'$. For P3 we must show that states ζ, ζ' have $\eta \Rightarrow^* \zeta$ and $\eta' \Rightarrow^* \zeta'$ and $\zeta \sim \zeta'$. If $\eta = \eta'$ we may let ζ, ζ' be η also. Now assume $\eta \neq \eta'$. Here at last we are forced to explicitly list all possibilities for ξCONTROL that allow more than one transition. Since the source program lacks explicit nondeterminism, this amounts essentially to listing all ξCONTROL possibilities with more than one node. Most of the cases are trivial and can be handled like Case 1 in the following partial case analysis.

Case 1: [ξCONTROL is {*entering* EVENTEST, *entering* ODDNO}]. We may assume that

ηCONTROL = {*entering* EVENTEST, *leaving* ODDNO} and

η'CONTROL = {p, *entering* ODDNO},

where p is either *entering* EVENYES or *entering* EVENNO, according to whether or not $\xi X[\xi I] > 0$. For variables other than CONTROL, η' is exactly like ξ. For variables other than CONTROL and J, η is exactly like ξ. Since J does not figure in the test $X[I] > 0$, this test can be performed in state η and has the same outcome as in state ξ: $\eta X[\eta I] > 0$ iff $\xi X[\xi I] > 0$. Thus we have a state ζ with $\eta \Rightarrow \zeta$ and

ζCONTROL = {p, *leaving* ODDNO};

$\zeta J = \eta J = \xi J + 2$;

ζ like ξ otherwise.

On the other hand, executing ODDNO from η' leads to ζ' with $\eta' \Rightarrow \zeta'$ and

ζ'CONTROL = {p, *leaving* ODDNO};

$\zeta' J = \eta' J + 2 = \xi J + 2$;

ζ' like ξ otherwise.

Thus $\zeta = \zeta'$. In short, the operations and tests involved in $\xi \Rightarrow \eta$ and $\xi \Rightarrow \eta'$ do not interfere with each other and can be performed in either order. This is the commutativity property stressed in [25] (see also [16, Definition 2.2]).

Case 2: [ξCONTROL is {*entering* EVENSEARCH, *entering* FINDODD}]. We may assume that

ηCONTROL = {*entering* EVENSEARCH, *leaving* FINDODD} and

η'CONTROL = {p, *entering* FINDODD},

where p is either *entering* EVENTEST or *leaving* EVENSEARCH, according to whether or not $\xi I < \xi$ODDTOP. For variables other than CONTROL, η' is exactly like ξ. For variables other than CONTROL and ODDTOP, η is exactly like ξ Since a change in ODDTOP might affect the results of the test, some care is necessary.

Case 2.1: $[\xi I < \xi$ODDTOP$]$. Then p is *entering* EVENTEST, and we have $\eta' \Rightarrow \theta'$ with

θ'CONTROL = {*entering* EVENTEST, *leaving* FINDODD};

θ'ODDTOP = $\eta' J = \xi J$;

θ' like ξ otherwise.

Case 2.1.1: $[\eta I < \eta$ODDTOP$]$. Then $\eta \Rightarrow \theta$ with

θCONTROL = {*entering* EVENTEST, *leaving* FINDODD};

θODDTOP = ηODDTOP = ξJ;

θ like ξ otherwise.

Letting ζ be θ and ζ' be θ will suffice.

Case 2.1.2: $[\neg\, \eta I < \eta$ODDTOP$]$. Then $\eta \Rightarrow \theta$ with

θCONTROL = {*leaving* EVENSEARCH, *leaving* FINDODD};

θODDTOP = ηODDTOP = ξJ;

θ like ξ otherwise.

By passing to the successor of FINDODD in ODDYES, executing the leave statement, and passing from {*leaving* EVENSEARCH, *leaving* ODDSEARCH} to {*leaving* SEARCH}, we get $\theta \Rightarrow^3 \zeta$ with

ζCONTROL = {*leaving* SEARCH};

ζODDTOP = ξJ;

ζ like ξ otherwise.

On the other hand, in θ' we may branch in EVENTEST to either EVENYES or EVENNO, perform the indicated operations, and so on.

Case 2.1.2.1: $[\theta' X[\theta' I] > 0]$. We may enter EVENYES, perform FINDEVEN, and leave EVENSEARCH. Leaving ODDSEARCH too and then SEARCH leads to $\theta' \Rightarrow^* \zeta'$ with

ζ'CONTROL = {*leaving* SEARCH};

ζ'ODDTOP = ξJ;

ζ'EVENTOP = ξI;

ζ' like ξ otherwise.

We will prove $\zeta \sim \zeta'$. In the definition of $\sim$ we must use (5.3.3) for the control state {*leaving* SEARCH}. We need

$$\min(\zeta\text{EVENTOP}, \zeta\text{ODDTOP}) = \min(\zeta'\text{EVENTOP}, \zeta'\text{ODDTOP}). \quad (1)$$

Using the inequalities presupposed by the present case, we calculate that

$$\min(\zeta\text{EVENTOP}, \zeta\text{ODDTOP}) = \min(\xi\text{EVENTOP}, \xi J)$$

where $\xi\text{EVENTOP} = m + 1$ and $\xi J \leq m$;

$$\min(\zeta'\text{EVENTOP}, \zeta'\text{ODDTOP}) = \min(\xi I, \xi J)$$

where $\xi J = \eta\text{ODDTOP} \leq \eta I = \xi I$.

Therefore both minima are ξJ and (1) follows. The sublemmas that EVENTOP = $M + 1$ and $J \leq M$ are true in ξ may be proved by the usual kind of inductive invariance argument, as in Lemma 3.4. Here we are concerned with computations that can lead to ξ rather than all computations.

Case 2.1.2.2: $[\neg\theta' X[\theta' I] > 0]$. We may increment I, leave EVENTEST, and return to EVENSEARCH, where we find that I has the value $\xi I + 2$ and ODDTOP has the value $\xi J = \eta\text{ODDTOP} \leq \eta I = \xi I$. Thus $I < \text{ODDTOP}$ is false and we may leave EVENSEARCH. Continuing from the place *leaving* FINDODD in θ'CONTROL, we find that $\theta' \Rightarrow^* \zeta'$ with

ζ'CONTROL = {*leaving* SEARCH};

ζ'ODDTOP = ξJ;

$\zeta' I = \xi I + 2$;

ζ' like ξ otherwise.

Since I is not relevant to $\sim$ in this control state, $\zeta \sim \zeta'$.

Case 2.2: $[\neg\xi I < \xi\text{ODDTOP}]$. Then p is *leaving* EVENSEARCH, and we have $\eta' \Rightarrow \theta'$ with

θ'CONTROL = {*leaving* EVENSEARCH, *leaving* FINDODD};

θ'ODDTOP = ξJ;

θ' like ξ otherwise.

Case 2.2.1: $[\eta I < \eta\text{ODDTOP}]$. By $\eta I = \xi I \geq \xi\text{ODDTOP} \geq \eta\text{ODDTOP}$, this cannot arise.

Case 2.2.2: $[\neg\eta I < \eta\text{ODDTOP}]$. Then $\eta \Rightarrow \theta$ with

θCONTROL = {*leaving* EVENSEARCH, *leaving* FINDODD};

θODDTOP = ξJ;

θ like ξ otherwise.

Letting ζ be θ and ζ' be θ' will suffice.

Case 3: [ξCONTROL is {*entering* FINDEVEN, *entering* ODDSEARCH}]. Similar to Case 2. This is the last nontrivial case in the verification of P3. □

Theorem 5.6. *For any array* $x[1:m]$ *of integers, all computations by* FINDPOS *with inputs x to X and m to M finish in final states with output z in Z such that*

$(z = 0$ and $x[h] \leq 0$ *for all* h *in* $\{1, ..., m\})$ *or*

$(z = x[k]$ *for* $k = min\ \{h \mid x[h] > 0\})$.

Proof. Apply Theorem 4.6 and Lemmas 5.1, 5.2, 5.4 and 5.5. □

6. Prospects for computer assistance

The subtleties of parallelism underscore the widely held view that neither a human nor a computer can be very proficient in the near future at proving correctness of large programs when working alone. The human becomes impatient and makes small logical mistakes that often reflect small bugs in the program. The computer lacks the insight to choose invariant assertions such as (2) in the proof of Lemma 3.4. An interactive system would allow the insightful human and the patient computer to accomplish much more together than either could alone. This is not the place to survey current work on computer assisted correctness proofs. We only wish to sketch the implications for this work of asynchronous parallelism in general and the Church–Rosser approach in particular. By exploding the number of control states, asynchronous parallelism provides a strong incentive to use several simple invariant assertions rather than one or two complex ones that branch according to the value of CONTROL. The Church–Rosser approach facilitates the use of simple invariants by deriving correctness from four lemmas.

Certification and optimization are intimately related [35, 40]. The correctness proof here applies analytic concepts originally developed for the sake of optimization. (Conversely, certification methods may apply to optimization.) The dominance relation [26, 41] and changes in variables along control flow paths [2, 26] were used in verifying assertions. Live variable analysis [15, 19] figured in the choice of the equivalence relation among interpreter states. (Additional mechanizable heuristics to aid in this choice could perhaps be developed. The problem is left open here.) Partly because of structured programming, the analysis could remain informal for our small example. In practice, computer assistance would be needed. Computers can presently do flow analysis rapidly and reliably for programs in languages similar to those considered in classical work on certification [27]. Extensions to practical procedural languages are under study [3, 38].

To prove Lemma 5.5 we had to verify the property P3 for each of 164 control states, but only two cases were nontrivial. It is clearly feasible for a computer to enumerate the control states, deal with the trivial ones, and assist a human in managing the lush trees of subcases that grow from the nontrivial cases.

In effect, we proved Lemma 5.2 by transforming **parbegin** and **parend** to **begin** and **end** in FINDPOS. This transformation is crucial to the correctness proofs in [5,

7]. The Church–Rosser approach is more flexible. Where convenient, we could prove weak partial correctness with the aid of transformations that insert coroutine linkages instead. For example, processes which can send each other messages or data via buffers could be so linked. The correctness proof for the buffer machinery (as in [9]) could then be kept separate from the proof of weak partial correctness. Indeed, we are free to prove finishing, weak partial correctness, admissibility, and the Church–Rosser property by any means that work. The example correctness proof exploited structured programming and flow analysis methods in order to establish the four lemmas in a manner amenable to computer assistance.

Note added in proof. With a modification of CONTROL it may be possible to extend Theorem 2.8 so as to allow recursion, thanks to a technical device for describing recursive control flow without jumps or macroexpansion. This *path tree* device will be explained in a revision of [38] to appear in *J. ACM* under the title "Data flow analysis for procedural languages."

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, Code optimization and finite Church–Rosser systems, in: *Design and Optimization of Compilers*, R. Rustin (ed.) (Prentice-Hall, Englewood Cliffs, N. J., 1972) 89–105.

[2] F. E. Allen, A basis for program optimization, in: *Information Processing* 71, *C. V. Freiman (ed.)* (North-Holland, Amsterdam, 1972) 385–390.

[3] F. E. Allen, Interprocedural data flow analysis, in: *Information Processing* 74, J. L. Rosenfeld (ed.) (North-Holland, Amsterdam, 1974) 398–402.

[4] E. A. Ashcroft, Proving assertions about parallel programs, Tech. Rept. CS–73–01, Dept. of Applied Analysis and Computer Science, Univ. of Waterloo, Waterloo, Ont. (1973).

[5] E. A. Ashcroft and Z. Manna, Formalization of properties of parallel programs, in: *Machine Intelligence* 6, B. Meltzer and D. Michie (eds.) (American Elsevier, New York, 1970) 17–41.

[6] A. Birman, On proving correctness of microprograms, *IBM J. Res. Develop.* 18 (1974) 250–266.

[7] J. M. Cadiou and J. J. Levy, Mechanizable proofs about parallel processes, in: *Proc. 14th Ann. IEEE Symp. on Switching and Automata Theor* (1973) 34–48.

[8] M. Dezani-Ciancaglini and M. Zacchi, Application of Church–Rosser properties to increase the parallelism and efficiency of algorithms, in: *Second Colloq. on Automata, Languages, and Programming*, Lecture Notes in Computer Science 14 (Springer, Berlin, 1974) 170–184.

[9] E. W. Dijkstra, A constructive approach to the problem of program correctness, *BIT* 8 (1968) 174–186.

[10] E. W. Dijkstra, Co-operating sequential processes, in: *Programming Languages*, F. Genuys (ed.) (Academic Press, New York, 1968) 43–112.

[11] R. W. Floyd, Assigning meanings to programs, *Proc. Symp. Appl. Math.* 19 (1967) 19–32.

[12] J. A. Goguen and J. W. Thatcher, Initial algebra semantics, *Proc. 15th Ann. IEEE Symp. on Switching and Automata Theory* (1974) 63–77.

[13] J. N. Gray, Locking, *Proc. ACM Conf. on Concurrent Systems and Parallel Computation* (1970).

[14] M. S. Hecht and J. D. Ullman, Flow graph reducibility, *SIAM J. Comput.* 1 (1972) 188–202.

[15] M. S. Hecht and J. D. Ullman, Analysis of a simple algorithm for global flow problems, *Proc. ACM Symp. on Princ. of Programming Languages* (1973) 207–217.

[16] R. M. Karp and R. E. Miller, Parallel program schemata, *J. Comput. System Sci.* 3 (1969) 147–195.

[17] R. M. Keller, A fundamental theorem of asynchronous parallel computation, *3rd Ann. Sagamore Conf.* (1974).

[18] R. M. Keller, On proving correctness of parallel programs, Rept. 159 (revised), Computer Sci. Lab., Princeton Univ. (April 1975).

[19] K. Kennedy, A global flow analysis algorithm, *Internl. J. Comput. Math.* **3** (A) (1971) 5–15.
[20] D. E. Knuth, Semantics of context-free languages, *Math. Systems Theory* **2** (1968) 127–145.
[21] S. R. Kosaraju, Analysis of structured programs, *J. Comput. System Sci.* **9** (1974) 232–255.
[22] H. Langmaack, On procedures as open subroutines II, *Acta Informat.* **3** (1974) 227–242.
[23] H. C. Lauer, Correctness in operating systems, Ph.D. Thesis, Computer Science Dept. Carnegie-Mellon University, Pittsburgh (September 1972).
[24] R. J. Lipton, Limitations of synchronization primitives with conditional branching and global variables, *Proc. 6th Ann. ACM Symp. on Theory of Computing* (1974) 230–241.
[25] R. J. Lipton, Reduction: a new method of proving properties of systems of processes, *Proc. 2nd ACM Symp. on Principles of Programming Languages* (1975) 78–86.
[26] E. S. Lowry and C. W. Medlock, Object code optimization, *Comm. ACM* **12** (1969) 13–22.
[27] Z. Manna, The correctness of programs, *J. Comput. System Sci.* **3** (1969) 119–127.
[28] Z. Manna, The correctness of nondeterministic programs, *Artificial Intelligence* **1** (1970) 1–26.
[29] Z. Manna, Mathematical theory of partial correctness, *J. Comput. System Sci.* **5** (1971) 239–253.
[30] R. Milner, An algebraic definition of simulation between programs, *Second Internat. Joint Conf. on Artificial Intelligence* (1971) 481–489.
[31] F. L. Morris, Advice on structuring compilers and proving them correct, *Proc. ACM Symp. on Princ. of Programming Languages* (1973) 144–152.
[32] P. Naur (ed.), Revised report on the algorithmic language ALGOL 60, *Comm. ACM* **6** (1963) 1–17.
[33] P. Naur, Proof of algorithms by general snapshots, *BIT* **6** (1966) 310–316.
[34] M. Nivat, On some families of languages related to the Dyck language, *Proc. 2nd ACM Symp. on Theory of Computing* (1970) 221–225.
[35] L. J. Osterweil and L. D. Fosdick, Data flow analysis as an aid in documentation, assertion generation, validation, and error detection, Report CU-CS-055-74. Computer Sci. Dept., Colorado Univ., Boulder (September 1974).
[36] W. W. Peterson, T. Kasami and N. Tokura, On the capabilities of while, repeat, and exit statements, *Comm. ACM* **16** (1973) 503–512.
[37] B. K. Rosen, Tree-manipulating systems and Church–Rosser theorems, *J. ACM* **20** (1973) 160–187.
[38] B. K. Rosen, Data flow analysis for recursive PL/I programs, IBM Research Rept. RC 5211, Yorktown Heights N.Y. (January 1975).
[39] R. Sethi, Testing for the Church–Rosser property, *J. ACM* **21** (1974) 671–679.
[40] R. L. Sites, Proving that computer programs terminate cleanly, Ph.D. Thesis, STAN-CS-74-418, Computer Science Dept., Stanford (May 1974).
[41] R. Tarjan, Finding dominators in directed graphs, *SIAM J. Comput.* **3** (1974) 62–89.
[42] W. A. Wulf, Programming without the GOTO, *Information Processing* 71, C. V. Freiman (ed.) (North-Holland, Amsterdam, 1972) 408–413.
[43] W. A. Wulf, A case against the GOTO, *SIGPLAN Notices* **7** (11) (November 1972) 63–69.